PCT WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 6 : G06F	A2	(11) International Publication Number: **WO 99/39254**
		(43) International Publication Date: 5 August 1999 (05.08.99)

(21) International Application Number: PCT/US99/02073

(22) International Filing Date: 29 January 1999 (29.01.99)

(30) Priority Data:
09/015,566 30 January 1998 (30.01.98) US

(71) Applicant: 3COM CORPORATION [US/US]; 5400 Bayfront Plaza, Santa Clara, CA 95052 (US).

(72) Inventors: SHAW, Richard, L.; 13425 South 2200 West, Riverton, UT 84065 (US). ADAMS, Phillip, M.; 1466 Chandler Drive, Salt Lake City, UT 84103 (US). MASON, Jack, L.; 21 South Powell Road, Elk Ridge, UT 84651 (US). GRAY, Jonathan, Dale; 2759 South Blair, Salt Lake City, UT 84115 (US). BULLOUGH, Jeffery, C.; 9918 South Dream Circle, South Jordan, UT 84095 (US). ROLLINS, Randy, C.; 7093 South 2350 West, West Jordan, UT 84084 (US). FEAGANS, Raymond, John; 13065 Avian Place, Nevada City, CA 95959 (US).

(74) Agents: JOHANSON, Kevin, K. et al.; Workman, Nydegger & Seeley, 1000 Eagle Gate Tower, 60 East South Temple, Salt Lake City, UT 84111 (US).

(81) Designated States: AU, CA, JP, European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE).

Published
Without international search report and to be republished upon receipt of that report.

(54) Title: SOFTWARE ARCHITECTURE FOR PROVIDING LOW LEVEL HARDWARE DEVICE DRIVERS FROM THE USER MODE UNDER MULTI–TASKING OPERATING SYSTEMS

(57) Abstract

A method and architecture for interfacing to a low level device driver in the user mode portion of an operating system having at least a user mode and a supervisor mode. The architecture includes a thin layer supervisor mode system interface driver for presenting a complete interface to a user application employing said low level device driver for executing user commands. The architecture also has a device routing driver portion also located in the supervisor portion of the operating system for routing between the thin layer supervisor mode system interface driver and the device driver located in the user mode portion of the operating system. The device drivers typically require minimal to no modifications as a device driver wrapper emulates an API environment for the device driver. The device driver wrapper facilitates relinking of the device driver to the device driver wrapper without requiring recoding.

FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AL	Albania	**ES**	Spain	**LS**	Lesotho	**SI**	Slovenia
AM	Armenia	**FI**	Finland	**LT**	Lithuania	**SK**	Slovakia
AT	Austria	**FR**	France	**LU**	Luxembourg	**SN**	Senegal
AU	Australia	**GA**	Gabon	**LV**	Latvia	**SZ**	Swaziland
AZ	Azerbaijan	**GB**	United Kingdom	**MC**	Monaco	**TD**	Chad
BA	Bosnia and Herzegovina	**GE**	Georgia	**MD**	Republic of Moldova	**TG**	Togo
BB	Barbados	**GH**	Ghana	**MG**	Madagascar	**TJ**	Tajikistan
BE	Belgium	**GN**	Guinea	**MK**	The former Yugoslav Republic of Macedonia	**TM**	Turkmenistan
BF	Burkina Faso	**GR**	Greece			**TR**	Turkey
BG	Bulgaria	**HU**	Hungary	**ML**	Mali	**TT**	Trinidad and Tobago
BJ	Benin	**IE**	Ireland	**MN**	Mongolia	**UA**	Ukraine
BR	Brazil	**IL**	Israel	**MR**	Mauritania	**UG**	Uganda
BY	Belarus	**IS**	Iceland	**MW**	Malawi	**US**	United States of America
CA	Canada	**IT**	Italy	**MX**	Mexico	**UZ**	Uzbekistan
CF	Central African Republic	**JP**	Japan	**NE**	Niger	**VN**	Viet Nam
CG	Congo	**KE**	Kenya	**NL**	Netherlands	**YU**	Yugoslavia
CH	Switzerland	**KG**	Kyrgyzstan	**NO**	Norway	**ZW**	Zimbabwe
CI	Côte d'Ivoire	**KP**	Democratic People's Republic of Korea	**NZ**	New Zealand		
CM	Cameroon			**PL**	Poland		
CN	China	**KR**	Republic of Korea	**PT**	Portugal		
CU	Cuba	**KZ**	Kazakstan	**RO**	Romania		
CZ	Czech Republic	**LC**	Saint Lucia	**RU**	Russian Federation		
DE	Germany	**LI**	Liechtenstein	**SD**	Sudan		
DK	Denmark	**LK**	Sri Lanka	**SE**	Sweden		
EE	Estonia	**LR**	Liberia	**SG**	Singapore		

SOFTWARE ARCHITECTURE FOR PROVIDING LOW LEVEL HARDWARE DEVICE DRIVERS FROM THE USER MODE UNDER MULTI-TASKING OPERATING SYSTEMS

BACKGROUND OF THE INVENTION

1. The Field of the Invention

This invention relates to hardware device drivers used in an operating system. More particularly, this invention relates to hardware device drivers operating from the user mode in an operating system.

2. Present State of the Art

Very early modem architectures placed a communication modem external to the personal computer. Such an external placement relied upon the computer's interface port such as a COM port of the personal computer to interface with the modem through which data information could be exchanged. The external modem contained all of the functionality necessary for performing traditional modem functions such as providing modulation and demodulation functionality for transmitted data as well as performing CODEC functionality required by the telephone network to which the modem was connected.

As technology advanced, both the personal computer and the modem functionality became more integrated. Such an integration enabled the modem functionality to be implemented on a computer card for physical placement within the personal computer. That is to say, the functionality once implemented in an external module was integrated onto a computer card or circuit board including the modulation functionality as well as the CODEC functionality. As the personal computer increased in both capability and performance, substantial functionality once performed in an external modem or in an integrated circuit board modem was able to be integrated into the personal computer. Such functionality primarily took the form of software functionality such as the ability to handle AT commands, and protocols (*i.e.*, V.42). Such an implementation is commonly known in the industry as a WINmodem implementation wherein the supervisor portion is performed on the personal computer. In such an architecture, the modulation device, typically a digital signal processor (DSP), performs the data COM functions while the rest of the supervisor portion is implemented on the personal computer or host and is implemented in a Windows application.

In modern traditional operating systems, functionality is partitioned into different

privilege levels. One such partitioning configuration employs a user mode and a supervisor mode for implementing the privilege partitioning structure. For operating systems utilizing popular x86 processors, the utilization of the x86 architecture to facilitate the privilege partitioning takes advantage of the x86 utilization of various privilege modes commonly known as rings. X86 architectures employ a four ring partitioning of which most operating systems utilize only two of the rings. Generally, ring 0 is utilized for implementing the supervisor mode of the operating system while a ring 3 privilege level is employed for implementing a user mode.

Traditional WINmodem architectures have been implemented consistent with design directives from operating system implementors in that WINmodem designers have complied with the generation and placement of hardware device drivers in a ring 0 environment wherein the supervisor mode is implemented. That is to say, software modules such as device drivers placed in the ring 0, or supervisor mode area of the operating system literally become part of the operating system. Such an incorporation simply extends the operating system to do something in addition to the original functionality. When large blocks of code, such as device drivers, are integrated into ring 0 or the supervisor mode area of the operating system, the opportunities for the operating system to become unstable and even seize or "crash" due to the integration of such foreign code into the highly tested kernel mode portion of the operating system greatly increases. Therefore, in such a WINmodem configuration or other like configuration the hardware device driver is integrated into the supervisor mode and becomes a supervisor mode driver with only minor functionality such as a CODEC implemented on an external card or integrated onto the mother board.

In a traditional WINmodem or host-based modem implementation, the functionality of the hierarchy as depicted in Figure 1 was employed. In the architecture, a traditional communication application 10 (*e.g.*, HyperTerminal, QuickLink, ProCom, NetScape) operating in a user mode communicates in the operating system, using a COMM API 12 to transition from a ring 3 privilege level to a ring 0 privilege level through a VCOMM virtual communication driver 14 in a Microsoft operating system.

A device driver such as Winmodem VxD device driver 16 resident within a supervisor mode interfaces to a virtual device driver such as VCOMM 14 and hardware such as hardware 18 which may take the form of a modem or other communication-type device and may further include video type devices.

In such an architecture, much of the functionality that was original on the external

board or module is now incorporated into a ring 0 or supervisor mode portion of the operating system. Such functionality in the case of a modem includes functions such as the data pump and optionally includes modulation functionality. Because of the substantial nature of these software components, the size of the supervisor mode portion of the operating system becomes extremely augmented by the inclusion of such code within the kernel. Because of the substantial nature of the code included within the kernel, and the generally less-thoroughly tested nature of third-party or aftermarket device drivers, an operating system generally becomes more unstable and susceptible to crashing. Furthermore, since such code is located within the supervisor mode or ring 0 portion of the architecture, a crash or misbehavior of such code nearly always leads to the complete termination of the present operating session. Conversely, applications operating in the user mode that display deleterious behavior, are terminated by the supervisor mode portion of the operating system when such behavior renders the user mode application inoperable.

Additionally, when software vendors generate portions of the software for inclusion within the kernel mode, there are additional and even onerous design requirements placed upon the software vendor. Such guidelines often inflict limitations on what a supervisor mode component of software may perform. Furthermore, such restrictive guidelines also render supervisor mode software very difficult to maintain due to the restriction and limitations of the functionality that may be performed. One exemplary limitation for performing such functionality in the kernel mode, is inherent in the architecture of x86 processors. For example, supervisor mode code utilizing features of the x86 architecture such as the floating point processor or the MMX processor are not fully reconstructed when context switching occurs. In an operating system switching between user and supervisor modes, the context or register values and other information such as stack pointers are preserved during a switch, thus enabling a process to resume a specific execution thread by reinstalling the register and other flag values when returning to the process. However, in the x86 architecture the floating point values and the MMX values are not preserved as part of a typical context switch task. Therefore, supervisor mode components utilizing these features become volatile when values in the floating point unit and MMX process are not preserved. While many supervisor mode components require sophisticated processing such as that capable of being performed by a floating point processor, instability of the operating system is extremely common.

Thus, it appears that there exists no present technique for providing low level hardware device drivers from the user mode of an operating system. Furthermore, there

does not currently exist techniques for enhancing the reliability of an operating system by placing low level device drivers in the user mode portion of an operating system to preclude or minimize the vulnerability of the operating system to ill- behaved software components. Therefore, a need exists for providing a method and apparatus for placing low level device drivers in the user mode portion of an operating system and facilitating direct interaction with the hardware directly therefrom.

SUMMARY

Placement of software components such as device drivers in the supervisor mode basically extends the operating system thereby providing additional opportunity for the operating system to fail. In contrast, placement of software components in the user mode portion of an operating system protects the kernel functions of the operating system. Furthermore, wayward software components in the user mode portion of an operating system generally only cause the termination of that particular user application and not the crashing of the entire system.

The present invention provides an architecture having a thin layer supervisor mode driver that appears to a standard user application as a device driver capable of servicing hardware directly from the supervisor mode. The present invention further comprises a routing driver component coupled to the thin layer device driver to communication with a device driver wrapper that provides a "supervisor mode-like" environment for a device driver. Because of the capabilities of the device driver wrapper, device drivers may be extracted from their supervisor mode linkage and relinked to the device driver wrapper without requiring recoding as is customary when porting software modules. Furthermore, a device driver may be reused between platforms once a device driver wrapper exists for each platform with the reusability of a device driver across multiple platforms being capable of being extended to a family of standard products that are interchangeable across platforms.

These features of the present invention will become more fully apparent from the following description and appended claims, or may be learned by the practice of the invention as set forth hereinafter.

BRIEF DESCRIPTION OF THE DRAWINGS

In order that the manner in which the above-recited and other advantages of the invention are obtained, a more particular description of the invention briefly described above will be rendered by reference to specific embodiments thereof which are illustrated in the appended drawings. Understanding that these drawings depict only typical embodiments of the invention and are not therefore to be considered to be limiting of its

scope, the invention will be described and explained with additional specificity and detail through the use of the accompanying drawings in which:

Figure 1 is a simplified architectural diagram of traditional communication modules and their operational locations, in accordance with prior art implementations;

Figure 2 is a simplified architectural diagram of the components and relationships for providing low level hardware drivers from the user mode, in accordance with the preferred embodiment of the present invention;

Figure 3 is a more detailed description of the functional interfaces between the various components for facilitating the relocation of a device driver to the user mode of an operating system, in accordance with the preferred embodiment of the present invention;

Figure 4 is a simplified depiction of functionality incorporated within the user mode to facilitate the placement of a device driver in the user mode of a partitioned operating system, in accordance with the preferred embodiment of the present invention;

Figure 5 is a simplified diagram of supervisor mode support drivers employed to facilitate the operation of a device driver in user mode, in accordance with the preferred embodiment of the present invention;

Figure 6 is a simplified block diagram of the components and relationships for providing low level hardware drivers from the user mode in a Windows NT, in accordance with an alternate embodiment of the present invention;

Figure 7 is a simplified depiction of functionality incorporated within the user mode to facilitate the placement of a device driver in the user mode of a Windows NT operating system, in accordance with an alternate embodiment of the present invention; and

Figure 8 is a simplified diagram of supervisor mode support drivers employed to facilitate the operation of a device driver in user mode in an embodiment employing a Windows NT operating system, in accordance with the preferred embodiment of the present invention.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

The present invention embodies within its scope both methods and an apparatus for enabling a user mode device driver to interact directly with an I/O port or hardware device without requiring the direct services of the supervisor mode. The present invention also provides a method and apparatus for placing traditional supervisor mode drivers in the user mode portion of the operating system thereby enabling the operating system to terminate a wayward user mode device driver without causing the entire

operating system to seize with a supervisor mode device driver exhibits aberrant behavior.

Figure 2 is a simplified block diagram of the architecture employing a thin supervisor mode system interface to portray to the operating system a compatible interface while routing commands and data to a device driver located in the user mode for executing and interacting directly with an I/O port or hardware without intermediary involvement by the supervisor mode. The present invention allows the operating system to see the system interface connections that are expected in the supervisor mode, while performing only minimal work in supervisor mode and placing the majority of functionality in the user mode where they are far less likely to cause catastrophic volatility to the operating system when problems occur in a device driver.

Actual testing of the present architecture has yielded test results showing that for a typical modem exchange between a communication application and a corresponding hardware device, execution of the transfer was performed at over a 95/5 duty cycle with the substantial majority of the time being consumed by execution in the user mode. The architecture of the present invention appears to the user and the user application such as a commercial-off-the-shelf communication application that the majority of the work is being traditionally performed by a device driver in supervisor mode when in fact the vast majority of the work is being performed in the user mode.

Typical operating systems generally operate in two different privilege modes, a first privilege mode such as user mode 26 and a second privilege mode such as supervisor mode 28. All user applications execute and operate in user mode 26. Typically, user mode applications include E-mail, word processing, and graphical programs such as games. In a typical architecture, the user mode cannot communicate directly with an I/O port or hardware. The user mode traditionally is restricted in the functions and processes that it can request the system to perform. For instance, one user application is prevented from interacting or interfering with the operation of another user application. The operating system in user mode attempts to keep each user application isolated to prevent them from being improperly behaved and incurring a negative impact on other user applications.

Supervisor mode 28 is the portion of the architecture where the operating system kernal resides. The incorporation of supervisor mode software modules such as device drivers extends the operating system, thus enabling it to perform additional specific functions. Typical supervisor mode functionality specifically includes interacting with I/O ports and hardware (e.g., video display, hard drive, CD ROM drive and floppy drive).

In supervisor mode 28, software modules can see all memory, I/O, interrupts and has access to all other components. Normally, user mode 26 has no such exposure or control.

A user application 30 generally operates in user mode 26, however, user application 30 needs the services that the operating system provides in the supervisor mode, and therefore, execution of user application 30 must cross between user mode 26 and supervisor mode 28 to obtain some of those services. In fact, user application 30 really can't perform rudimentary functionality such as obtaining a file off of the disk or storing information to a disk without the services of supervisor mode 28.

User application 30 and a system interface driver 34 communicate using Application Programming Interfaces (APIs) 32, a portion of which have been defined in the present invention and a standard set used by the operating system. APIs 32 provide fundamental command interface capable of passing the appropriate information and getting the operating system or the driver which is an extension of the operating system in supervisor mode 28 to take that action. For instance, user application 30 may request to open a file named ABC.PPT, which is essentially performed by putting in a command with the data that is affiliated with that command followed by the issuance of the specific API. The specific API is comprised of a certain number of predefined parameters that it needs. Therefore, if a block of data is targeted for storage, then one of the parameters for that API is a pointer to the data.

In a traditional architecture implementation, a device driver would reside in the supervisor mode, either a standard device driver or a custom device driver for a specific type of I/O port or hardware and carries out the specific data processing and appropriate control of the hardware as necessary. In the present invention, instead of a traditional device driver residing in supervisor mode 28, the supervisor portion, system interface driver 34 and device router driver 44, are much thinner and leaner with the majority of the functionality located in the user mode portion of the invention. The supervisor mode portion basically presents the minimal interface entry points that the particular function requires by the operating systems. Basic operating system functionality and compatibility dictate the a particular kind of device must present a certain interface to the operating system defining the specifics for interacting with the operating system.

In one implementation of the present invention utilizing a Microsoft® operating system, there is actually a layer provided by the operating system that exists between system interface driver 34 and user application 30. This layer, when for example, in user mode 26 calls into the operating system using a communications API and is responsible for all the resources in the system. It performs other functions such as (i) checking to

verify if a device is in the system to handle a specific type of a request, (ii) which component needs to respond in supervisor mode, and (iii) the interface with which the operating system needs to interact. Therefore, system interface driver 34 provides a thin driver interface to provide the specific set of entry points for the driver that the operating system requires. None of the device driver functionality is performed in the supervisor mode, although the operating system and user application perceive it as being consistent with the more traditional supervisor mode location of the device driver.

In a typical scenario, user application 30 calls in with an API to the operating system with a device name (e.g., Com1, Com2) that the operating system recognizes. User application 30 calls through API 32 which in turn determines that, for example, Com1 corresponds with system interface driver 34. System interface driver 34 is further comprised of at least one base class 36 which defines a unique requirement for how interface driver 34 "hooks" to the operating system with the respective entry points for use by the operating system when interacting. For example, if a disk driver interface is to be implemented, base class 36 would be referring to a disk driver base class that provides insight and understanding about the functionality the operating system will need to employ (i.e., it will need to perform functions such as open driver, driver reads, driver writes, driver status, etc.). To facilitate functionality, each kind of device driver will have unique types of commands that will be received from the operating system, therefore, each kind of driver will need a unique base class connection to make the architecture function properly. A specific example of base class 36 is implemented in the Microsoft® Windows NT 4.0 operating system under the name of a port class and employs approximately 21 specific actions that the operating system is going to expect the driver to perform for proper interaction (e.g., understanding of data structures, command structures) and the work that is required out of each one through the architecture to actually get the work done. In the preferred embodiment, a generic base class is defined as opposed to defining base class 36 as, for example, a SCSI or IDE class.

In a flow of the present invention, when a call comes from user application 30, for example, to open a device, the call comes through API 32 through the operating system into system interface driver 34. Prior to the call for a specific service, the architecture passes through an initialization phase wherein the operating system is booting or instantiating itself the first time. In such a process, an operating system registry 38 is consulted, for example, to determine how many device drivers are needed for a session. Following the read of operating system registry 38, the operating system instantiates an instance of the necessary interfaces and the links to device router 44 based

on that information in operating system registry 38. Therefore, for example, if only one device will be supported in a session, the requisite data would be reflected in operating system registry 38 and would be read by system interface driver 34 as instantiation was underway while identifying the quantity of base classes to create within system interface driver 34, and the quantity of connections to establish between system interface driver 34 and device router driver 44. Therefore, in an example where one device will be supported, only a single instantiation is performed and if a multiple number of devices are to be supported then a multiple number of instantiations will be performed.

A configuration program 42 is executed as the operating system boots. Configuration program 42 contains sufficient knowledge about the operating system registry 38, the operating system itself and potential hardware or devices or device drivers that could be used in the system. Configuration program 42 also contains functionality to determine through operating system registry 38 the presence of devices for the session or devices that have been recently inserted, which is yet further notified through an operating system configuration manager 40, when, for example, a PCMCIA card or a CardBus card has been inserted during operation. Operating system configuration manager 40 recognizes that the new device has been inserted, works with the operating system in supervisor mode to determine who is intended to support that device using an identification string that identifies each hardware piece in operating system registry 38 once the device is initially installed. When the device is inserted or is found present by operating system configuration manager 40, configuration program 42 in user mode 26 establishes a connection between itself and operating system configuration manager 40.

When the device is recognized by operating system configuration manager 40, operating system configuration manager 40 calls through interface 52 into user mode 26 and notifies configuration program 42 that a device is present. Configuration program 42 registers as the object that is going to take care of the device and registers itself with operating system configuration manager 40 requesting that it be notified if devices are inserted.

Next, configuration program 42 identifies the device, works through operating system registry 38 to store information regarding the settings to use for the hardware, driver or drivers for that session, writes that information into operating system registry 38 and sends a message 43 to a device driver wrapper 46 to notify the architecture that a new device is present requesting support. Device driver wrapper 46 then sends a message through interface 56 to go from user mode 26 down to device router driver 44 stating that a new device is present and the location where information about the new

device is located in operating system registry 38. Therefore, once configuration program 42 has stored the information in operating system registry 38 and sends the message to device driver wrapper 46, the functionality of configuration program 42 is complete.

Device driver wrapper 46 sends a message to device router driver 44 which then instantiates the appropriate handlers 60, 66 and 68 and device router driver 44 then sends a message to system interface driver 34 for notification that an additional instantiation is needed. Both system interface driver 34 and device router driver 44 read the information stored in operating system registry 38 by configuration program 42 and configure themselves accordingly to allocate system resources such as memory or IO space, interrupts and other services that are only feasibly performed by a device driver in supervisor mode. Such "supervisor mode-only" services are then set up by system interface driver 34. To this point, an instantiation of base class 36 that is configured to the parameters that the operating system deduced in conjunction with operating system configuration manager 40 and configuration program 42 are complete. Next, device driver wrapper 46 loads device driver 48 for use by the particular target device such as an I/O port or hardware. That is to say, device driver wrapper 46 loads device driver 48 and then instantiates all the remaining threads that constitute the rest of interface or threads 56.

Following the loading of device driver 48 by device driver wrapper 46, device driver wrapper 46 makes the connection through threads 56 to device router driver 44 and device router driver 44 reads operating system registry 38 and instantiates a link name 62 which is a unique link name for every device supported and a unique driver name for every driver to be supported. For example, this provides a method wherein a port open command comes in from user application 30 and system interface driver 34 knows through operating system registry 38 that COM1 is really COMm calling down from user mode 26 into supervisor mode 28 ending up in system interface driver 34 and instantiating one of base classes 36 thereby behaving similar to an alias for COM1. For example (see Figure 8), one user application 30 decides to operate as COM3 to system interface driver 34' but is known as COMSMDevice 0 in one instance. Because of the unique name, device router driver 44" from operating system registry 38 knows that in this particular case system interface driver 34" is to behave as a regular modem and that is, for example, link driver_0 in device router driver 44". Therefore, it knows to send the calls through threads 56".

Furthermore in the present example (Figure 8), if the user makes a different selection in an exemplary user application to select, for example, operation in a cellular mode such as originating a GSM call with the same hardware, operating continues through COM3, therefore, system interface driver 34" still presents itself as COM3 and is known in the system as COM3 but now operates with device router driver 44" to link to driver_1 instead of driver_0. Linking to driver_1 operates through another interface of threads 56"' to device driver wrapper for driver_1 instead of driver_0. Therefore, a device driver wrapper is present for each device driver and also a set of threads 56 exists between device router driver 44" and each device driver wrapper. As far as the operating system is concerned and the user application is concerned, it appears as if it is still COM3. That is to say, internally the system interface driver presents the exact same face to the world but the device router driver steers it to and from supervisor mode 28 through a different link driver object 60 (Figure2). In summary, link driver objects 60,66 and 68 can steer routing from one to the other and when device driver_0 48 is hooked to link driver 60 which could be a standard modem and device driver_1 72 with its wrapper 70 at user mode with its set of threads like 56 could be working through link driver 66 and be a GSM file and by simply changing the information in the registry through an application or an applet, as known by some operating systems, different functionality may be created through the same virtual pipe through the system. As far as the user is concerned, the hardware remains the same, the user application remains the same but internally to the architecture of the present invention, it is routed differently to a different device driver running in ring 3 to achieve a different result.

Figure 3 depicts a more detailed look at interfaces between system interface driver 34, device router driver 44 and device driver wrapper 46 (Figure 2). Thread 54 is a single thread between system interface driver 34 and device router driver 44. Thread 54 is a single thread per device that is being supported. Therefore, if there is one device driver or one hardware device or I/O port, then there will be one thread 54. This single thread provides a single entry point where a command or a value that directs, for example, an interrupt request, an open or a close, a read or a write or a status. Therefore, interface 90 which in one embodiment employs an internal IOCTL command common to Microsoft® operating systems is changed depending upon the number of things that need to be performed which is influenced by the kind of driver being supported by the architecture. Thread 54 is created in the operating system when system interface driver 34 instantiates itself.

Threads 56 between device router driver 44 and individual device driver wrappers

46, 70 and 74 are individual threads: interrupt, device open, device close, device read, device write and device get status are all unique threads. Figure 5 is a diagram of a single device instance. If there are multiple devices then there would be multiple instances of each of these. It should be reiterated that in Figure 3, thread 54 and threads 56 are not symmetrical. In thread 54, one thread performs all of the functionality depending on which function is requested. For example, suppose the user application has requested an open command. A thread is created in the operating system between system interface driver 34 and device router driver 44. System interface driver 34 calls into the device IO control entry point at device router driver 44 with thread 54 which is basically a request to have the operation performed. Device router driver 44 pends thread 54 and since it is the only thread that is between the two devices, no other calls from the operating system or user application can come through until that one is completed.

Device router driver 44 then takes the information from the device IO control call and that pended thread 54 and completes device router driver's 44 request via threads 56 DeviceOpen thread for that device to which the command was directed. A simplified description suffices to state that the DeviceOpen thread to the device driver, runs in device driver wrapper 46 and device driver 48 to perform all the necessary functionality to open the device or I/O port. Once the device is opened, device driver wrapper 46 issues to device router driver 44 on that same device open thread asking it to run that one again and device router driver 44 pends that thread. that, and we'll talk about the rest of that here in a moment.

When data comes back from the device driver in user mode from the open command, the open operation is complete and device router driver 44 pends the new thread from the open that was just performed. Since the process is not yet ready to be repeated, it is not necessary to perform an open yet and the thread is not run. The data is then put in the data structure that the operating system specifically wants to see and the single thread 54 is completed which then executes the data by moving it up to the user application through the original open call through interface 32.

It should be clarified that thread 54 when it is not performing is complete and therefore not executing. Also, during initialization, when device driver wrapper 46 first instantiates, it links itself to device router driver 44 by establishing all the threads that comprise threads 56 and by calling through using the device IO control, for example, from user mode 26 to supervisor mode 28 and in the exemplary case of a communication (COM) device, 23 individual threads per device get instantiated between device driver wrapper 46 and device router driver 44.

As the initialization process proceeds, each thread is received and pended by device router driver 44 which means that once the components of the present invention are initialized and the user application is yet to issue requests, thread 54 between system interface driver 34 and device router driver 44 is complete and not functioning, while all of the 23 threads of threads 56 from device router driver 44 in supervisor mode 28 to device driver wrapper 46 located in user mode 26 are pended. Pending threads provides performance enhancement since if one were to simply call from user mode 26 from an application down to supervisor mode 28, a context switch would be required every time, which as those skilled in the art appreciate, requires substantial time to perform due to the delay in storing register and pointer values and loading other register and pointer values.

A second problem also occurs in that every time one crosses the boundary between user mode and supervisor mode, the operating system takes that opportunity to service any pending high priority events. Because device drivers need to operate in substantially real time, the introduction of significant delays introduced between calls becomes unacceptable. Therefore, by initiating and pending the individual threads of threads 56 performance is greatly enhanced by foregoing context switching. Those skilled in the art appreciate the setting of a thread to "complete" starts the thread executing again and a mode transition and hence a context switch is not necessary.

Additionally, context switching between modes introduces problems inherent in the x86 architecture in that co-processor registers are not preserved. By placing the device driver in user mode, the operating system processes the switching, for example in multitasking, and therefore forgoes the x86 shortcomings. Furthermore, in the present invention, the threads are prioritized relative to the operating system.

Figure 4 depicts the composition of the device driver and the components of the device driver wrapper, in accordance with the preferred embodiment of the present invention. Prior discussion relating to device driver wrapper 46 have depicted it as a monolithic entity when in the preferred embodiment of the present invention, device driver wrapper 46 is more incitefully depicted as a composition of functional modules which greatly enhance the reusability and minimize the quantum of code that must be customized between implementations on various hardware and operating system platforms. Device driver wrapper 46 may also be viewed as a frame or component that facilitates the relocation of a device driver from the supervisor mode to the user mode with minimal alterations to the device driver.

In Figure 4, the architecture of the present invention is depicted such that the device driver originating in a supervisor mode may be relinked with device driver wrapper 46 and as far as device driver 48 knows, it is still operating as usual. Component 86 functions as a driver interface API to user mode API emulation/real time emulation applications. As device driver 48 is concerned, it perceives itself as operating in supervisor mode. What the present invention has done is to put a wrapper around device driver 48 and enabled it to continue to operate as it was originally designed. In order to provide such an environment for device driver 48, component 86 provides a real time emulation that is created using standard operating system calls and functions to emulate that environment for device driver 48. Additionally, component 114 provides an operating system for user mode API set which is basically the standard operating system for user mode APIs. In Figure 4, component 86 is illustrated above component 114 because of the API expanding effect of component 86. Device driver wrapper 46 provides all of the interfaces required by device driver 48 when operative in the supervisor mode and facilitates the extraction of a device driver previously designed to operate in supervisor mode and the subsequent placement of the device driver into the user mode by plugging the device driver into the device driver wrapper.

For example, in one embodiment, device driver wrapper 46 emulates as far as device driver 48 is concerned a system interface layer in supervisor mode. Therefore, as far as device driver 48 is concerned, it has hooked itself just like it does in supervisor mode to that system interface layer for its specific functionality. For a COM driver that would be VCOMM in a Microsoft operating system. So as far as the device driver is concerned, it perceives that it has connected itself to VCOMM when in reality it has connected itself to the device driver wrapper. Because of this emulated environment provided by device driver wrapper 46, device driver 48 may be utilized in its entirety without stripping or reworking portions of the device driver. In the preferred embodiment of the present invention, component 114 is part of the operating system while component 86 could alternatively be merged with device driver wrapper 46.

Figure 5 is a simplified block diagram of the architecture for multiple user applications interfacing with multiple device drivers. Multiple instances of thread 54 are depicted by thread 55 and 61. Likewise, device router driver 44' illustrates multiple device driver base classes 60, 66 and 68 while system interface driver 34' illustrates multiple driver base classes as well. Figure 5 further illustrates that each device has its own interrupt, unique port addresses and separate read and write data cues and threads.

Figure 6 depicts a simplified block diagram for implementing the present invention in a Microsoft NT® 4.x and WinModem architecture environment using functions and terminology consistent therewith. Therefore, instead of being a generic user application 30, a COM application would be created using NT specific definitions such as a port class rather than a base class. Additionally, in Figure 6, the device driver is designed to integrate into VCOMM or unimodem.

Figure 7 is a simplified block diagram of an implementation of the present invention within a Microsoft NT 4.0 operating system environment, in accordance with the preferred embodiment of the present invention. In Figure 7, a series of modular protocol modules are presented which include functionality such as fax, DTE, v.42, v.80, etc. which may be reused across multiple platforms. As described above, the present architecture provides a partitioned structure such that reuse is maximized requiring only the minimal amount of code to be rewritten when porting or upgrading operating systems. Device driver 48' is depicted as being taken from a supervisor role, which while not a requirement, certainly enhances the reusability of legacy drivers.

Likewise, device driver wrapper 46' provides a Win32 compatible API environment for device driver 48'. One of the few portions of the user mode driver created by the architecture of the present invention is the hardware specific interfacing portion for directly controlling the hardware. In Figure 7, the SDPI DRV represents that portion of the code that is specific to unique hardware. STPI DRV is an API chosen to adopt into a WinModem and soft modem implementations but a LAN driver would not have one of those.

The present invention may be embodied in other specific forms without departing from its spirit or essential characteristics. The described embodiments are to be considered in all respect only as illustrative and not restrictive. The scope of the invention is, therefore, indicated by the appended claims rather than by the foregoing description. All changes which come within the meaning and range of equivalency of the claims are to be embraced within their scope.

What is claimed is:

1. In an operating system having at least a user mode and a supervisor mode for execution of a user application on a computer, a method for driving an Input/Output (I/O) port from a device driver operating in said user mode, said method comprising the steps of:

a) receiving at a system interface driver operating in said supervisor mode an I/O port command from said user application transferred via an operating system API and a communication resource of said operating system, said system interface driver further providing a compatible interface to said communication resource of said operating system;

b) communicating said I/O port command via an internal device IO control command between said system interface driver and a device router driver also operating in said supervisor mode;

c) routing said I/O port command to a device driver resident within said user mode;

d) said device driver resident within said user mode obtaining direct privileged access to said supervisor mode of said operating system; and

e) said device driver resident within said user mode presenting said I/O port command as processed within said device driver to said I/O port.

2. The method for driving an Input/Output (I/O) port from a device driver operating in said user mode, as recited in claim 1, further comprising the steps of:

a) establishing at least one thread between said system interface driver and said device router driver for use in said communication step to send said internal device IO control command therebetween; and

b) establishing at least one thread between said device router driver and said device driver resident within said user mode for use in said routing step to send said I/O port command to said device driver.

3. The method for driving an Input/Output (I/O) port from a device driver operating in said user mode, as recited in claim 2, wherein said establishing steps comprise the steps of:

a) establishing said at least one thread between said system interface driver and said device router driver;

b) pending said at least one thread between said system interface driver and said device router driver;

c) establishing said at least one thread between said device router driver and said device driver resident within said user mode; and

d) pending said at least one thread between said device router driver and said device driver resident within said user mode.

4. The method for driving an Input/Output (I/O) port from a device driver operating in said user mode, as recited in claim 1, wherein said routing said I/O port command to a device driver resident within said user mode step comprises the steps of:

a) routing said I/O port command to a device driver wrapper resident within said user mode, said device driver wrapper providing an operating system API environment of user mode APIs to facilitate relocating said device driver from said supervisor mode to said user mode; and

b) said device driver wrapper issuing one of said user mode APIs to said device driver corresponding to said I/O port command.

5. The method for driving an Input/Output (I/O) port from a device driver operating in said user mode, as recited in claim 4, wherein operating system API environment of said device driver wrapper further comprises a means for emulating a driver interface to user mode API.

6. The method for driving an Input/Output (I/O) port from a device driver operating in said user mode, as recited in claim 5, wherein said obtaining direct privileged access to said supervisor mode step comprises the step of issuing one of said operating system APIs to request supervisor privileges of said supervisor mode from said device driver wrapper within said user mode.

7. In an operating system having at least a user mode and a supervisor mode for execution of a user application on a computer, a method for interfacing with an I/O port from a device driver operating in said user mode, said method comprising the steps of:

a) in a thin-layer supervisor mode system interface driver, presenting an interface to a communication resource of said operating system to receive from said user application via an operating system API and said communication service an I/O port command and wherein said communication service of said operating system perceives said system interface driver as comprising a device driver traditionally resident within said supervisor mode and capable of directly interfacing with said I/O port;

b) routing said I/O port command to said device driver resident within said user mode;

c) said device driver resident within said user mode obtaining direct privileged access to said supervisor mode of said operating system; and

d) said device driver resident within said user mode presenting said I/O port command as processed within said device driver to said I/O port.

8. The method for interfacing with an I/O port from a device driver operating in said user mode, as recited in claim 7, wherein said routing said I/O port command to said device driver resident within said user mode step comprises the steps of:

a) receiving at said system interface driver said I/O port command from said user application transferred via an operating system API; and

b) communicating said I/O port command via an internal device IO control command between said system interface driver and a device router driver also operating in said supervisor mode.

9. The method for interfacing with an I/O port from a device driver operating in said user mode, as recited in claim 8, further comprising the steps of:

a) establishing at least one thread between said system interface driver and said device router driver for use in said communication step to send said internal device IO control command therebetween; and

b) establishing at least one thread between said device router driver and said device driver resident within said user mode for use in said routing step to send said I/O port command to said device driver.

10. The method for interfacing with an I/O port from a device driver operating in said user mode, as recited in claim 9, wherein said establishing steps further comprise the steps of:

a) establishing said at least one thread between said system interface driver and said device router driver for use in transferring data between said I/O port and said user application;

b) pending said at least one thread between said system interface driver and said device router driver;

c) establishing said at least one thread between said device router driver and said device driver resident within said user mode for use in transferring data between said I/O port and said user application; and

d) pending said at least one thread between said device router driver and said device driver resident within said user mode.

11. The method for interfacing with an I/O port from a device driver operating in said user mode, as recited in claim 7, wherein said routing said I/O port command to a device driver resident within said user mode step comprises the steps of:

a) routing said I/O port command to a device driver wrapper resident within said user mode, said device driver wrapper providing an operating system API environment of user mode APIs to facilitate relocating said device driver from said supervisor mode to said user mode; and

b) said device driver wrapper issuing one of said user mode APIs to said device driver corresponding to said I/O port command.

12. The method for interfacing with an I/O port from a device driver operating in said user mode, as recited in claim 11, wherein operating system API environment of said device driver wrapper further comprises a means for emulating a driver interface to user mode API.

13. The method for interfacing with an I/O port from a device driver operating in said user mode, as recited in claim 12, wherein said obtaining direct privileged access to said supervisor mode step comprises the step of issuing one of said operating system APIs to request supervisor privileges of said supervisor mode from said device driver wrapper within said user mode.

14. In an operating system having at least a user mode and a supervisor mode for execution of a user application on a computer, a structure to enable said user application to interact via a device driver operating in said user mode with an I/O port as directed in an I/O port command from said user application, comprising:

a) a system interface driver for operating in supervisor mode, said system interface driver having minimal interface entry points required by said operating system and further appearing to said operating system as if said device driver is resident within said supervisor mode;

b) a device router driver operably coupled to said system interface driver and also resident in said supervisor mode to communicate said I/O port command between said system interface driver and said device driver and to route said I/O port command to said I/O port via said device driver; and

c) a device driver wrapper resident within said user mode and operably coupled to said device router driver to provide an operating system API environment of user mode APIs to facilitate relocating said device driver from said supervisor mode to said user mode, said device driver wrapper further capable of emulating a driver interface to user mode API.

15. The structure to enable said user application to interact via a device driver operating in said user mode with an I/O port as directed in an I/O port command from said user application, as recited in claim 14, wherein said system interface driver further

comprises at least one base class to define requirements of said system interface driver and how said system interface driver interfaces with said operating system.

16. A computer-readable medium for driving an Input/Output (I/O) port from a device driver operating in a user mode of an operating system having at least said user mode and a supervisor mode for execution of a user application on a computer, said computer-readable medium having computer-executable instructions for performing steps comprising:

a) receiving at a system interface driver operating in said supervisor mode an I/O port command from said user application transferred via an operating system API and a communication resource of said operating system, said system interface driver further providing a compatible interface to said communication resource of said operating system;

b) communicating said I/O port command via an internal device IO control command between said system interface driver and a device router driver also operating in said supervisor mode;

c) routing said I/O port command to a device driver resident within said user mode;

d) said device driver resident within said user mode obtaining direct privileged access to said supervisor mode of said operating system; and

e) said device driver resident within said user mode presenting said I/O port command as processed within said device driver to said I/O port.

17. The computer-readable medium of claim 16, having further computer executable instructions for performing the steps of:

a) establishing at least one thread between said system interface driver and said device router driver for use in said communication step to send said internal device IO control command therebetween; and

b) establishing at least one thread between said device router driver and said device driver resident within said user mode for use in said routing step to send said I/O port command to said device driver.

18. The computer-readable medium of claim 17, wherein said computer executable instructions for performing said establishing steps, comprise computer executable instructions for performing the steps of:

a) establishing said at least one thread between said system interface driver and said device router driver;

b) pending said at least one thread between said system interface driver and said device router driver;

c) establishing said at least one thread between said device router driver and said device driver resident within said user mode; and

d) pending said at least one thread between said device router driver and said device driver resident within said user mode.

19. The computer-readable medium of claim 16, wherein said computer executable instructions for performing said routing said I/O port command step comprise computer executable instructions for performing the steps of:

a) routing said I/O port command to a device driver wrapper resident within said user mode, said device driver wrapper providing an operating system API environment of user mode APIs to facilitate relocating said device driver from said supervisor mode to said user mode; and

b) said device driver wrapper issuing one of said user mode APIs to said device driver corresponding to said I/O port command.

20. The computer-readable medium of claim 19, wherein said computer executable instructions for said operating system API environment of said device driver wrapper further comprises computer executable instructions for implementing a means for emulating a driver interface to user mode API.

21. The computer-readable medium of claim 20, wherein said computer executable instructions for performing the step of obtaining direct privileged access to said supervisor mode comprises computer executable instructions for performing the step of issuing one of said operating system APIs to request supervisor privileges of said supervisor mode from said device driver wrapper within said user mode.

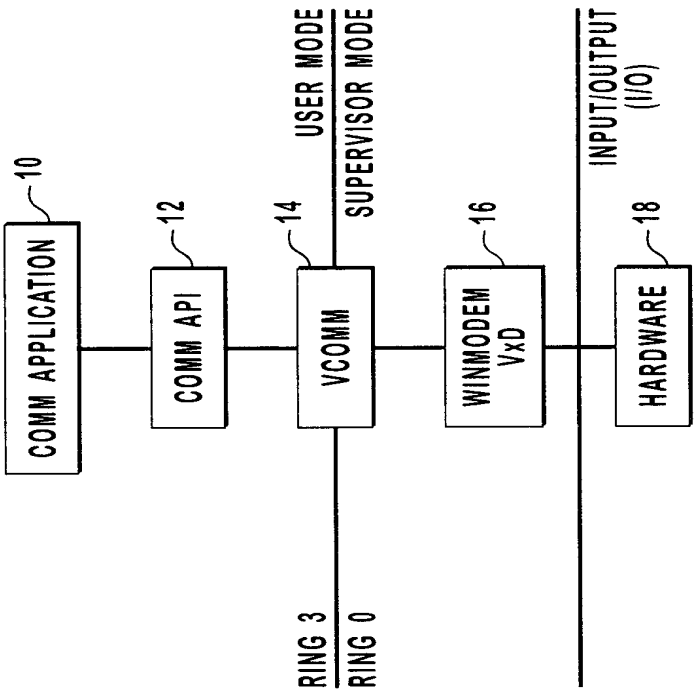

FIG. 1
(PRIOR ART)

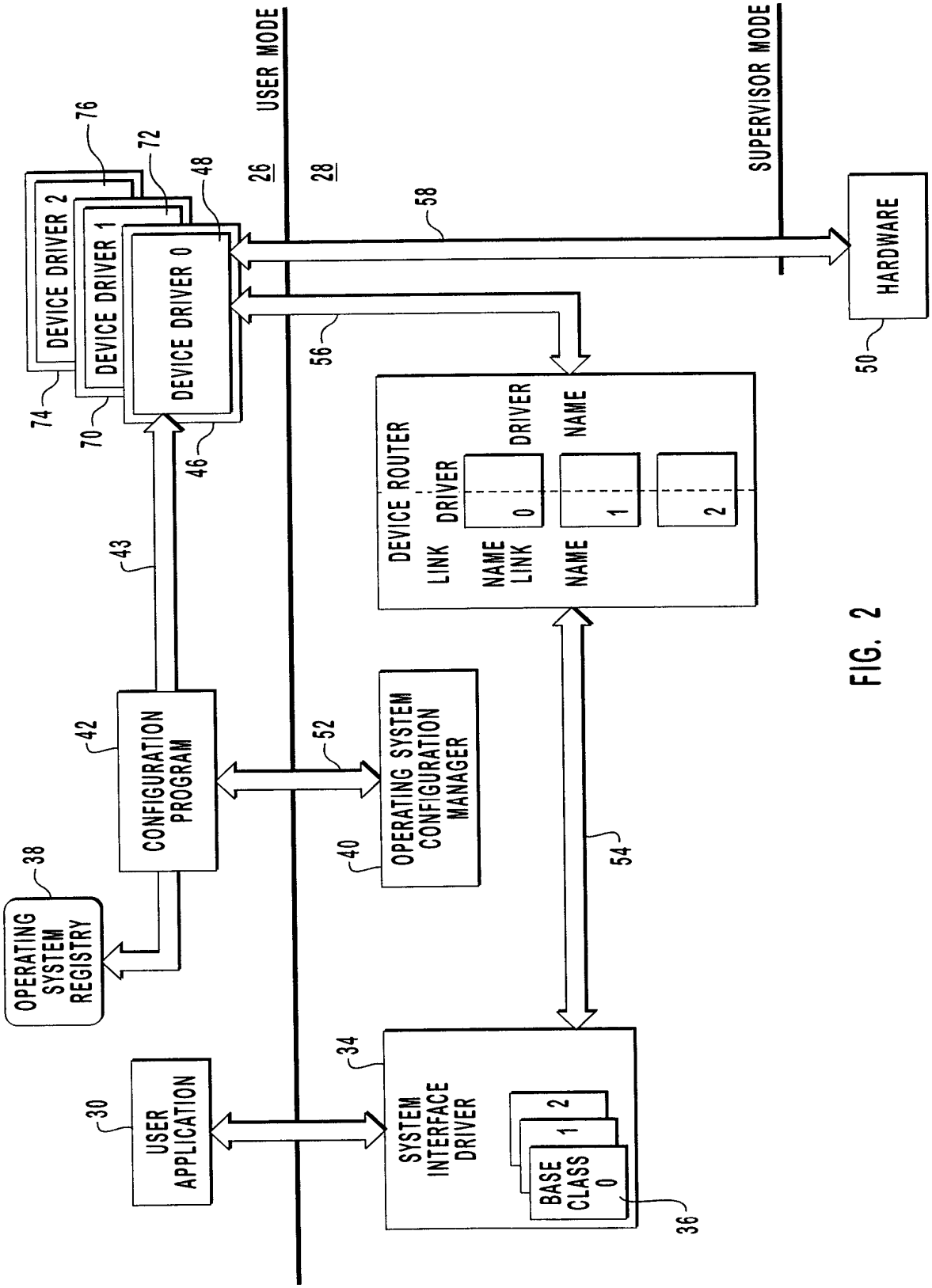
USER MODE
SUPERVISOR MODE
26
28
58
HARDWARE
50
DEVICE DRIVER 2
DEVICE DRIVER 1
DEVICE DRIVER 0
76
72
48
74
70
46
56
DEVICE ROUTER
LINK
DRIVER
NAME
NAME LINK
NAME
0
1
2
43
CONFIGURATION PROGRAM
42
52
OPERATING SYSTEM CONFIGURATION MANAGER
40
54
38
OPERATING SYSTEM REGISTRY
30
USER APPLICATION
34
SYSTEM INTERFACE DRIVER
BASE CLASS
0
1
2
36

FIG. 2

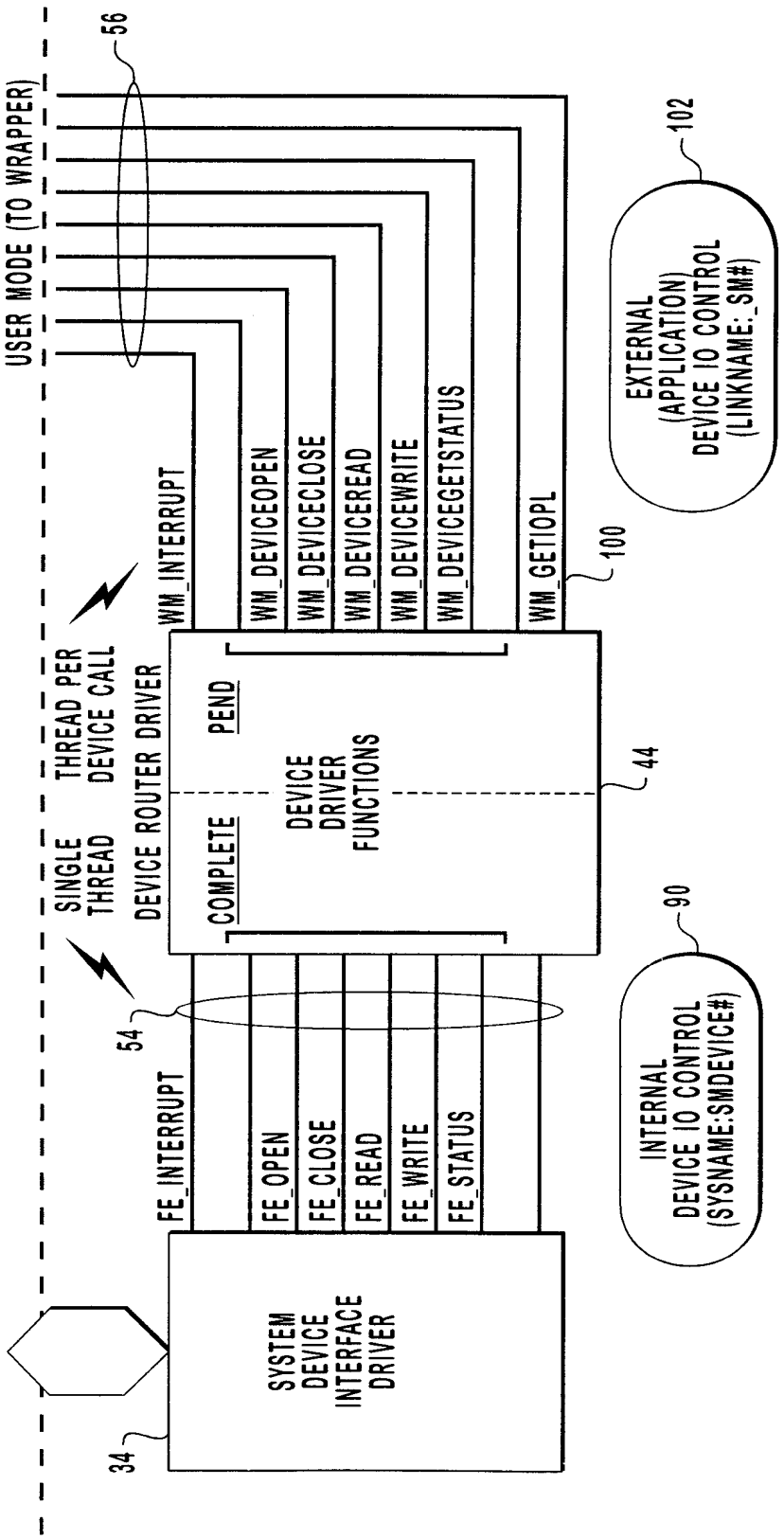
USER MODE (TO WRAPPER)
56
WM_INTERRUPT
WM_DEVICEOPEN
WM_DEVICECLOSE
WM_DEVICEREAD
WM_DEVICEWRITE
WM_DEVICEGETSTATUS
WM_GETIOPL
100
102
EXTERNAL (APPLICATION) DEVICE IO CONTROL (LINKNAME:_SM#)
SINGLE THREAD
THREAD PER DEVICE CALL
DEVICE ROUTER DRIVER
PEND
COMPLETE
DEVICE DRIVER FUNCTIONS
44
54
FE_INTERRUPT
FE_OPEN
FE_CLOSE
FE_READ
FE_WRITE
FE_STATUS
90
INTERNAL DEVICE IO CONTROL (SYSNAME:SMDEVICE#)
SYSTEM DEVICE INTERFACE DRIVER
34

FIG. 3

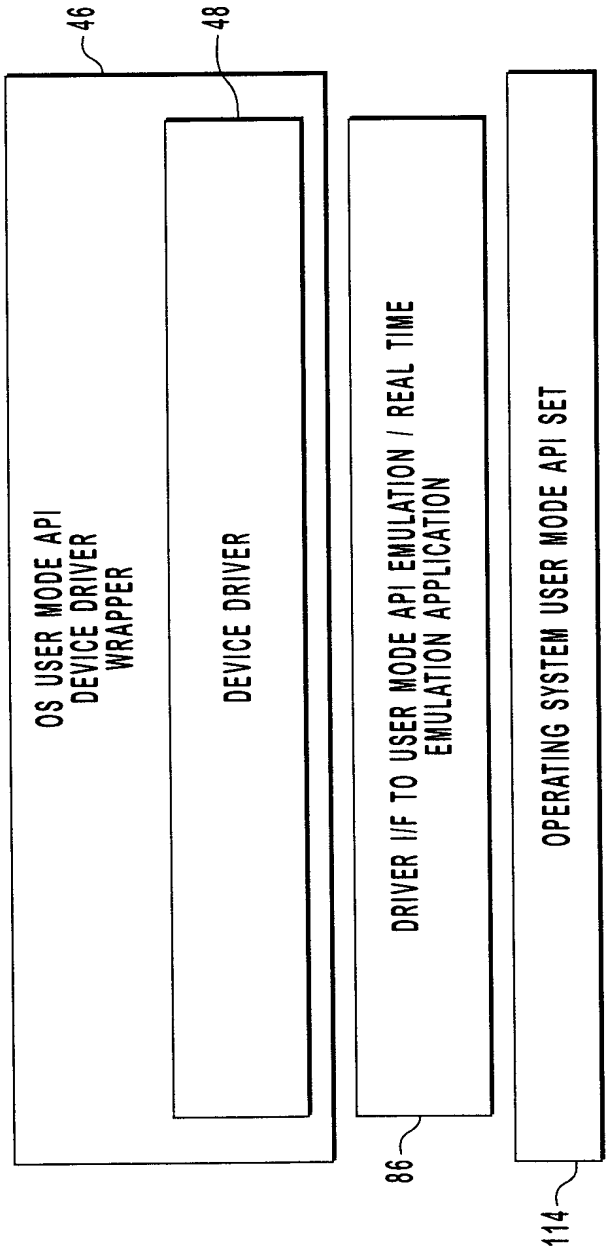
46
48
OS USER MODE API
DEVICE DRIVER
WRAPPER
DEVICE DRIVER
DRIVER I/F TO USER MODE API EMULATION / REAL TIME
EMULATION APPLICATION
OPERATING SYSTEM USER MODE API SET
86
114

FIG. 4

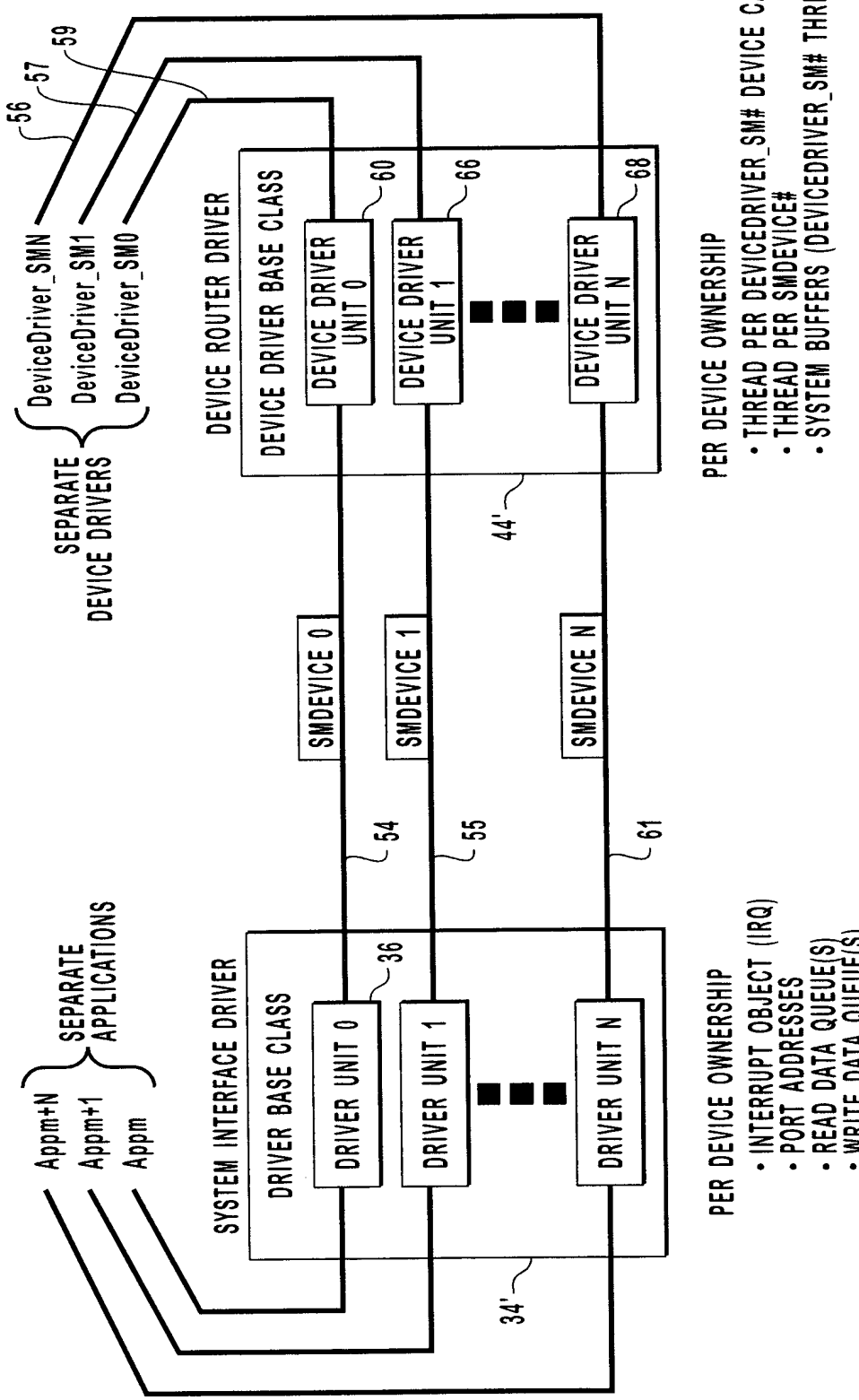
DeviceDriver_SMN
DeviceDriver_SM1
DeviceDriver_SM0
SEPARATE DEVICE DRIVERS
56
57
59
DEVICE ROUTER DRIVER
DEVICE DRIVER BASE CLASS
DEVICE DRIVER UNIT 0
DEVICE DRIVER UNIT 1
DEVICE DRIVER UNIT N
60
66
68
44'
PER DEVICE OWNERSHIP
• THREAD PER DEVICEDRIVER_SM# DEVICE CALL
• THREAD PER SMDEVICE#
• SYSTEM BUFFERS (DEVICEDRIVER_SM# THREADS)
SMDEVICE 0
SMDEVICE 1
SMDEVICE N
54
55
61
Appm+N
Appm+1
Appm
SEPARATE APPLICATIONS
SYSTEM INTERFACE DRIVER
DRIVER BASE CLASS
DRIVER UNIT 0
DRIVER UNIT 1
DRIVER UNIT N
36
34'
PER DEVICE OWNERSHIP
• INTERRUPT OBJECT (IRQ)
• PORT ADDRESSES
• READ DATA QUEUE(S)
• WRITE DATA QUEUE(S)
• READ THREAD QUEUE
• WRITE THREAD QUEUE

FIG. 5

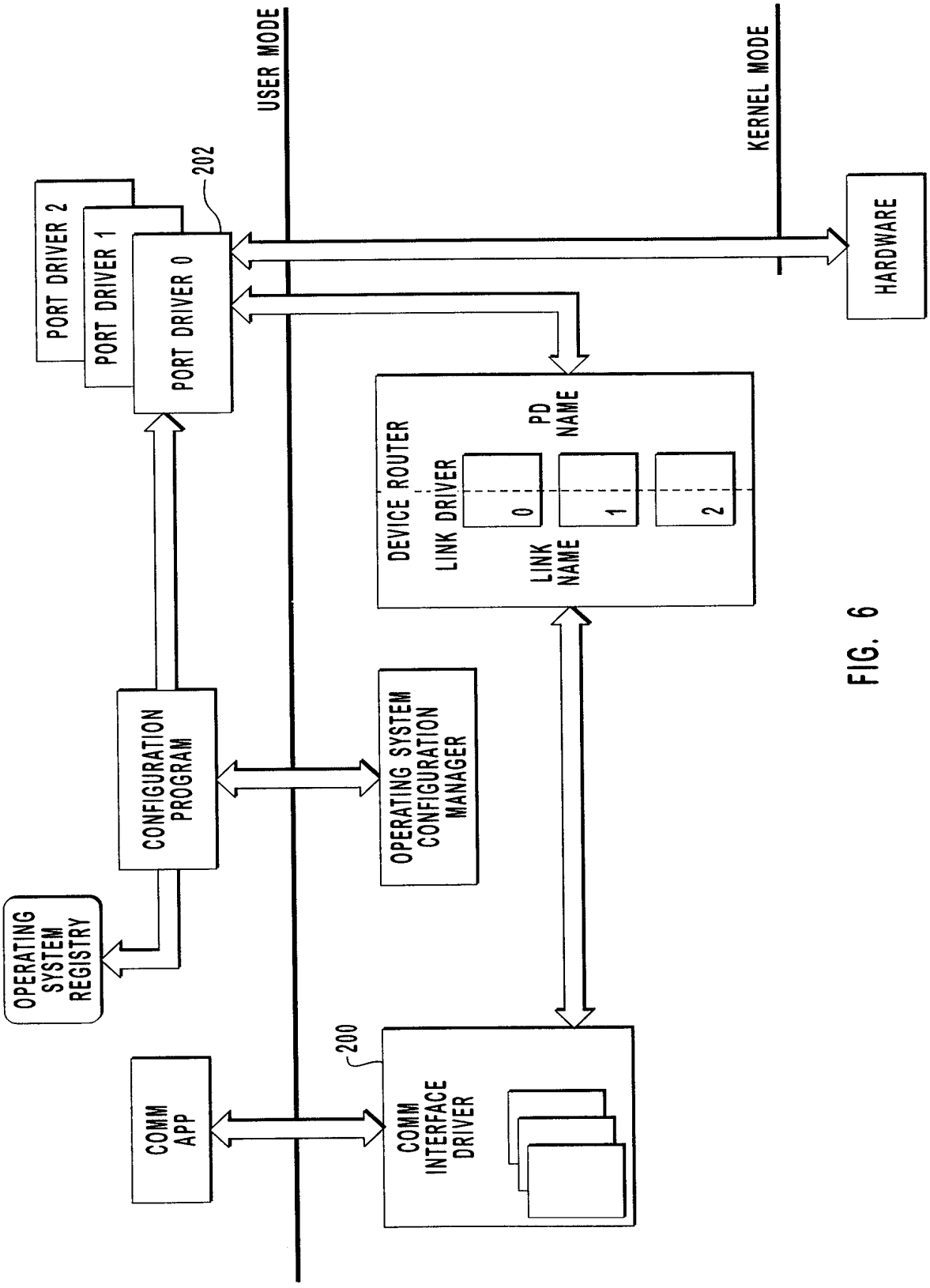
USER MODE
KERNEL MODE
202
PORT DRIVER 2
PORT DRIVER 1
PORT DRIVER 0
HARDWARE
DEVICE ROUTER
LINK DRIVER
PD NAME
LINK NAME
0
1
2
CONFIGURATION PROGRAM
OPERATING SYSTEM CONFIGURATION MANAGER
OPERATING SYSTEM REGISTRY
200
COMM APP
COMM INTERFACE DRIVER

FIG. 6

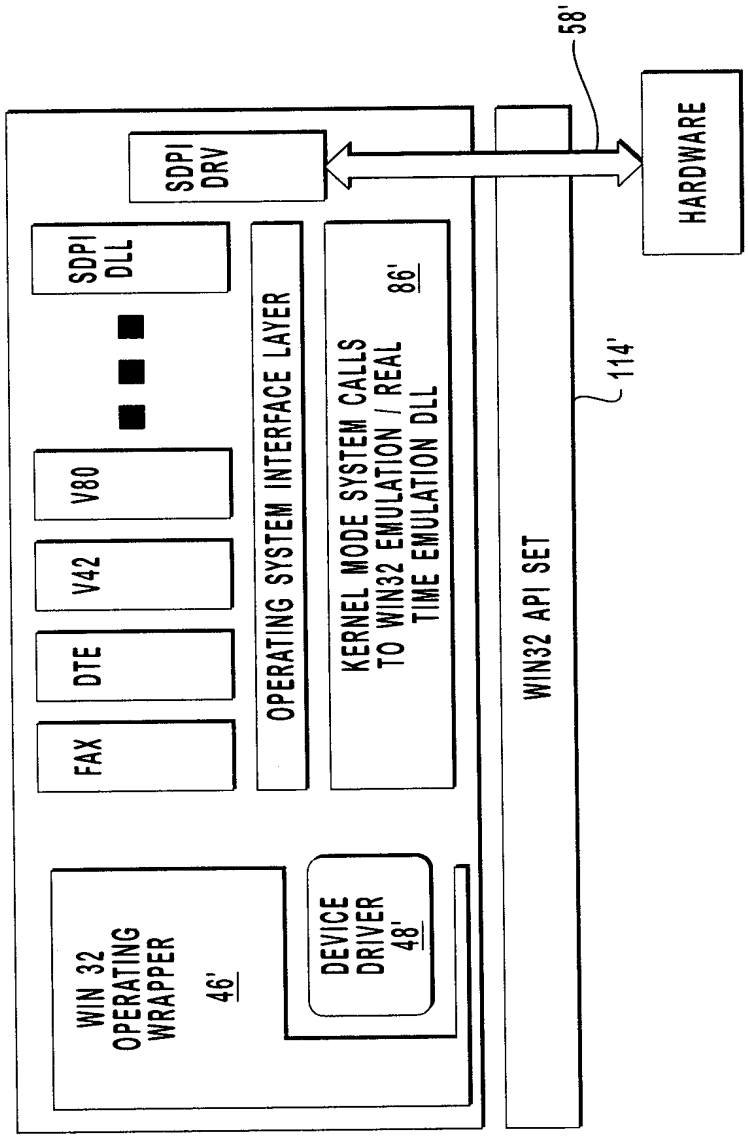
58'
HARDWARE
SDPI DRV
SDPI DLL
V80
V42
DTE
FAX
OPERATING SYSTEM INTERFACE LAYER
KERNEL MODE SYSTEM CALLS TO WIN32 EMULATION / REAL TIME EMULATION DLL
86'
114'
WIN32 API SET
WIN 32 OPERATING WRAPPER
46'
DEVICE DRIVER
48'

FIG. 7

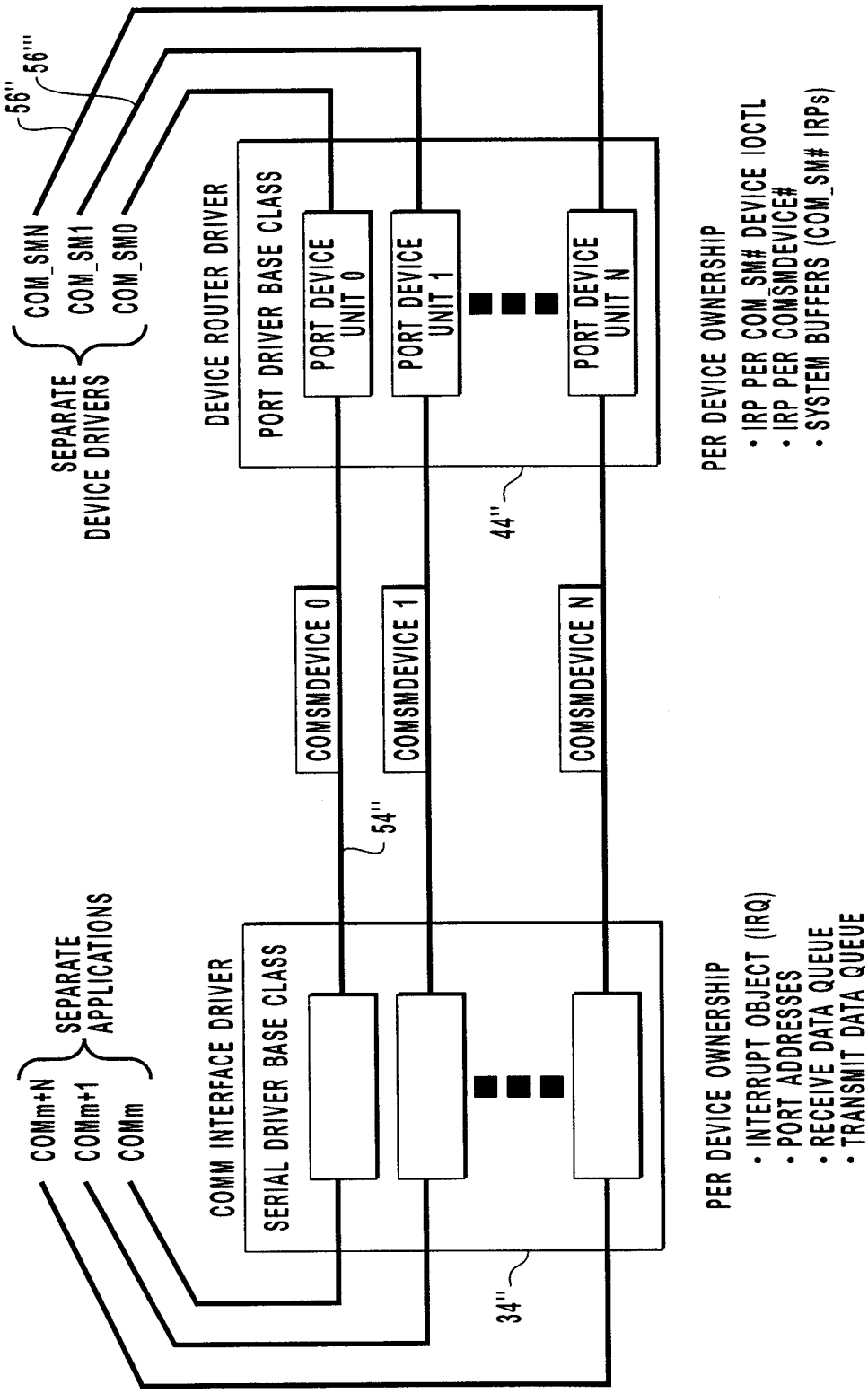
56''
56'''
COM_SMN
COM_SM1
COM_SM0
SEPARATE DEVICE DRIVERS
DEVICE ROUTER DRIVER
PORT DRIVER BASE CLASS
PORT DEVICE UNIT 0
PORT DEVICE UNIT 1
PORT DEVICE UNIT N
44''
PER DEVICE OWNERSHIP
• IRP PER COM SM# DEVICE IOCTL
• IRP PER COMSMDEVICE#
• SYSTEM BUFFERS (COM_SM# IRPs)
COMSMDEVICE 0
COMSMDEVICE 1
COMSMDEVICE N
54''
COMm+N
COMm+1
COMm
SEPARATE APPLICATIONS
COMM INTERFACE DRIVER
SERIAL DRIVER BASE CLASS
34''
PER DEVICE OWNERSHIP
• INTERRUPT OBJECT (IRQ)
• PORT ADDRESSES
• RECEIVE DATA QUEUE
• TRANSMIT DATA QUEUE
• READ IRP QUEUE
• WRITE IRP QUEUE

FIG. 8